

Scalable Instruction Set Simulator for Thousand-core Architectures Running on GPGPUs

Shivani Raghav, Martino Ruggiero († *), David Atienza
ESL - EPFL, Lausanne, CH.(†)
{shivani.raghav, martino.ruggiero, david.atienza}@epfl.ch

Christian Pinto, Andrea Marongiu, Luca Benini
DEIS - University of Bologna, Bologna, IT.(*)
{christian.pinto, a.marongiu, luca.benini}@unibo.it

ABSTRACT

Simulators are still the primary tools for development and performance evaluation of applications running on massively parallel architectures. However, current virtual platforms are not able to tackle the complexity issues introduced by 1000-core future scenarios. We present a fast and accurate simulation framework targeting extremely large parallel systems by specifically taking advantage of the inherent potential processing parallelism available in modern GPGPUs.

KEYWORDS: ISS, manycore, GPGPU, CUDA.

1. INTRODUCTION AND RELATED WORK

Despite the research effort devoted to continuously create higher performance processing systems, current methodologies for computing systems design and applications development are still based on simulation in both high performance computing (HPC) [11] and embedded system domains [15].

Future architectures will expose a massive battery of parallel very-simple processors and on-chip memories connected through a network-on-chip, which speed is more than hundred times faster than the off-chip one [3]. It is clear that current virtual platforms will not be able to tackle these future scenarios, because they suffer problems of either performance or scalability issues. Sequential simulators are quite accurate [4][14][2], but as the complexity of the simulated platform increases the

simulation time gets unreasonably large. Another method to address simulation duration is to make use of parallel simulation [13][7][12][5], but they require multiple processing nodes to increase the simulation rate.

During last years, the development of computer technology brought an unprecedented performance increase along with modern general-purpose GPU. They provide both scalable computation power and flexibility, and they have already been adopted for many computational intensive applications. Thanks to GPGPUs, the last five years were marked by the propagation of PC clusters based on such manycores leading to inexpensive solutions in high performance computing for a wide community. However, in order to obtain the highest performances on such a machine, the programmer has to write programs that best exploit the hardware architecture. None of the current simulators takes advantage of the computational power provided by modern GPGPUs.

The main contribution of this paper is the development of fast simulation design method and environment targeting extremely large parallel systems by specifically taking advantage of the inherent potential processing parallelism available in modern GPGPUs. We developed a new simulation technology to deploy a parallel simulator for 1000-core system on top of GPGPUs.

In the design of our simulation environment, we targeted two main application scenarios, namely high performance computing and embedded system platforms. For each of them we have implemented a different instruction set simulator (ISS) compliant to a different instruction set architecture (ISA). Considering the HPC domain, the IA-32 (x86) ISA has been developed, while the ARM ISA has been selected for the embedded system scenario. The

design and development of each of these ISSes have been particularly optimized for the modern GPGPU architectures.

2. ISS IMPLEMENTATIONS

In this section we describe the implementation of two multi-core processor models written in C++ and CUDA which simulate the ARM and IA-32 (x86) instruction sets. These ISS are meant to be in the future integrated within a complete architectural simulation framework, capable of simulating hundreds of cores, private instruction and data caches interconnected through a network (NOC, shared bus). At its present status of development, the purpose of our ISS is the fast execution of input instruction stream which sufficiently simulates the programmer visible state of the processors.

At the heart of this simulator, there is a small loop that repeatedly fetches loaded object code of target application for each of the simulated cores. Similar to the operation on the hardware, the instruction byte is fetched, decoded and executed at the run time. Each core updates their simulated registers files and program counter until the program is finished. For all instructions requiring memory references, the traces are sent to input/output buffers which provide a generic interface towards memory. This provides a flexible communication infrastructure in case the core simulator needs to be plugged to some other architectural component simulator (e.g. the cache).

Presently no such interaction is available, and thus memory operations are managed from within the ISS itself. One physical GPU thread is used to simulate one target machine processor. To model many cores, we used several parallel GPU threads. Each core has been assigned its own context structure which represents one CPU and its current state. Initially host (CPU) memory is allocated for a context structure that represents one core. Host program initializes the members of this structure and copies the data to the global device memory. When modeling the simulator for N cores, the global memory contains an array of N context structures whose fields constitute the register file, program counter and other core status flags.

2.1. ARM ISA Simulation

At the moment, our ARM ISS is capable of executing a representative subset of the ARM ISA. The Thumb mode is currently not supported. The simulation is decomposed into three main functional blocks: fetch, decode and execute. When implementing this functional model on top of the CUDA execution model several practical issues arise. GPU architecture imposes a common fetch step for

all of the threads belonging to a given warp. This in turn impacts the performance of the parallel program, since execution of threads fetching different instructions gets serialized.

2.2. Instruction Decoding

The ARM ISA leverages fixed length 32-bit instructions, thus making it straightforward to identify a set of 10 bits which allows decoding an instruction within a single step. These bits are used to index a 1024-entry Look-Up Table (LUT), thus immediately retrieving the opcode which univocally identifies the instruction to be executed (see Figure 1). The ARM ISA supports much less instructions than 1024 and all of the necessary opcodes could be stored in a very small data structure. Nonetheless we allow an increased memory footprint for the LUT ($1024 \text{ entries} \times \text{sizeof}(\text{unsigned short int}) = 2\text{KB}$) to preserve the simplicity of the indexing mechanism.

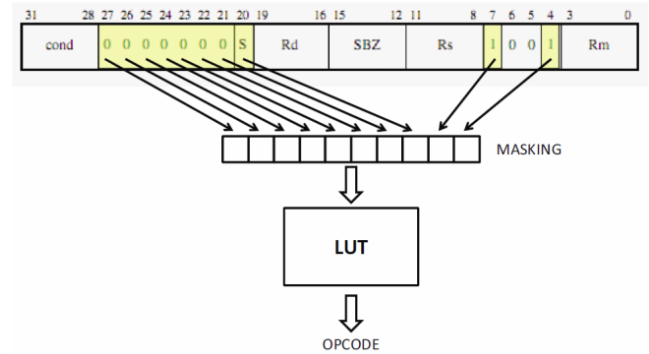


Figure 1. Instruction Decoding

Since it is very difficult to avoid sparse accesses to the LUT (due to processors fetching different program instructions), accessing it from the global memory would not be easily done under memory coalescing and would thus be inefficient. The LUT is statically declared and contains read-only data. To reduce its access cost we can take advantage of the texture memory. Texture memory operates as a cache between the global memory and the device. An object living in the global memory can be declared as cacheable by binding its declaration to a texture reference. Since our LUT is entirely contained within the texture memory (16KB) we only pay the cost for global memory accesses upon cache miss events at the beginning of the program. Besides retrieving the opcode, in this step we also extract from the instruction all of the fields necessary to its execution, such as source/destination operands, immediates and flags. Executing these decoding steps even in case the instruction does not require it is more performance-

efficient than conditionally differentiating the execution path.

2.3. Instruction Execution

In this step the previously extracted opcode and operands are used to simulate the target instruction semantics. Prior to instruction execution processor status flags are checked to determine whether to actually execute the instruction or not (e.g. after a compare instruction). In case the test is not passed a NOP instruction is executed.

Finally, the actual instruction execution is modeled within a switch/case construct. This is translated from the CUDA compiler into a series of conditional branches, which are taken depending on the decoded instruction. This point is the most critical to performance. In SPMD-like parallel computation – where each processor executes the same instructions on different data sets – CUDA threads are allowed to execute concurrently. In the worst case, however, on MIMD task-based parallel applications each processor may take a different branch, thus resulting in complete serialization of the entire switch construct execution.

To identify the boundaries within which programs may perform under this implementation we present in Sec. 5 an extensive set of experiments which consider both synthetic workloads aimed at reproducing worst and best cases, and real programs.

2.4. CPU Context Allocation

Given the criticality of memory accesses and their impact on simulation performance, it is of the utmost importance to carefully design the layout of simulated cores' execution contexts in memory. Contexts are represented with 16 general-purpose registers, a status register plus a auxiliary registers used for exception handling, or to signal the end of execution.

Contexts can be easily represented with a single data structure such as a matrix. One dimension sweeps through the available registers, the other identifies a simulated core. Due to the frequent accesses performed by every program to its execution context, it is beneficial to place this data structure in the low latency shared memory rather than accessing it from the global memory. This requires explicit copy operations, whose cost can be minimized if we design the matrix layout according to its access pattern. Since matrices are laid in memory in row-major order, having each thread read a distinct row leads to separate transactions, as shown in Fig. 2. On the contrary, scanning matrices in column-major order, and thus having separate

threads in a half-warp access the same memory block minimizes the number of necessary transactions for the copy operation, as shown in Fig. 3. To enable coalesced accesses it is also important to ensure that accesses are aligned to a 64-Byte boundary.

Based on the above considerations, we represent contexts in memory with a matrix similar to that sketched in Fig. 4. Padding is employed whenever the number of simulated cores is not an integer multiple of the size of a half-warp (16), and it is left to a specific CUDA API function

```
cudaMallocPitch((void **) &context.registers, &pitch,
               sizeof(unsigned int) * num_cores, REG_COUNT);
```

where pitch represents the matrix width in bytes. Upon execution (i.e. when the simulation kernel is launched) each core copies its own context in shared memory through its identifier (idx)

```
shared_registers[REG_ID] = global_registers[REG_ID
      * pitch / (sizeof(int)) + idx]
```

A similar copyout scheme is employed at the end of the kernel execution to restore the contexts in global memory.

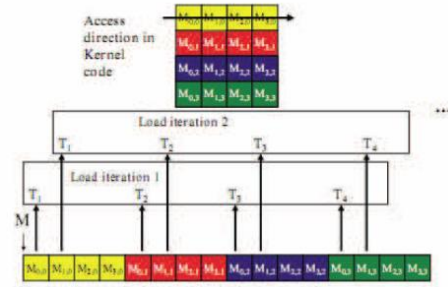


Figure 2. Scanning Matrices Row-major Leads to Memory Bandwidth Wastage

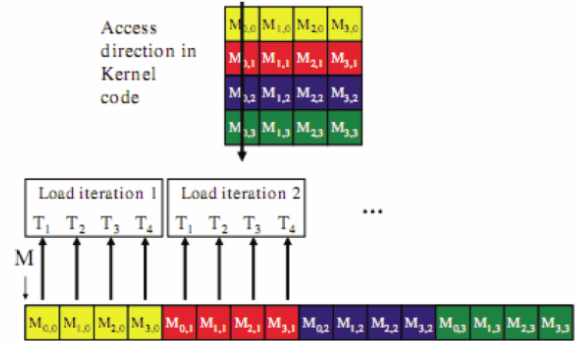


Figure 3. Scanning Matrices Column-major Minimizes Memory Transactions

2.5. x86 ISA Simulation

As the majority of high computing systems in the server and desktop market are either x86 instructions based or capable of emulating x86 code efficiently, we implemented a simulator which can handle Intel IA-32 instruction set architecture. The simulator takes the native machine code and executes it using the simplest pipeline of fetch, decode and execute. At the present stage of its development, we support all general purpose instructions and some of the feature as interrupt and exception handling are under development. Contexts are represented by an eight 32 bit general purpose registers, six segment registers, program status and control (EFLAGS) register and instruction pointer (EIP). We use memory coalescing

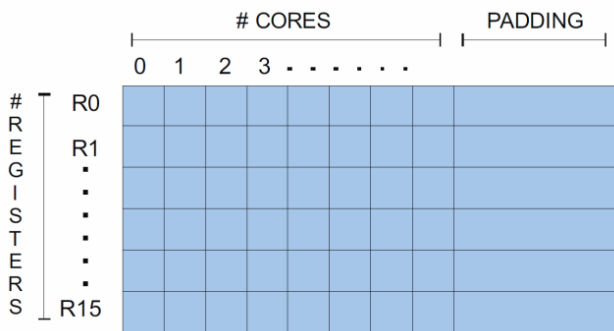


Figure 4. A Matrix Representing Execution Contexts on Simulated Cores

techniques for performance gain as already described in the previous section.

x86-32 is a CISC architecture, which employs complex decoding logic. Instructions have a variable length and are not aligned in the instruction cache, thus requiring multiple stages for decoding. The block diagram in Figure 5 shows the multiple steps of the decoding process. A single operation can be represented by multiple opcodes and the operands are determined in ways that are not well patterned. Unlike our ARM simulator, which has a common fetch and decode step for all instructions, this compromises the concurrency of CUDA threads. Indeed threads simulating different instructions branch to different functions depending on the operation parsed from the fetched opcode. To parse the instruction from the opcode byte we leverage a switch/case statement. All byte codes with a common operation call a function which further fetches and decodes the next subsequent bytes in a pipelined fashion. This involves unpacking ModR/M byte, determining operand size, addressing modes and implementing the effective address computation. When the information regarding the operand has been collected

it is then passed to short functions for simulation of target instruction operation. In a few special cases, prefix bytes are present that modify the following byte codes. Since most of the instructions that have prefixes (segment prefixes, address-size prefix, and the lock prefix), are not supported we have limited ourselves to only two. One is a word-size prefix (0x66) and the other is a two-byte opcode prefix (0x0f). Functions that execute them return special values that are then accumulated into an internal set of flags which then affect the decoding of subsequent bytes.

This implementation leverages conditional branches at several stages of the decoding step, thus implying performance penalties. This happens in particular for task-based parallel applications, where each processor may take a different branch and lead to massive warp serialization. We are considering two design methodologies for future versions of our simulator to reduce this performance loss. First, with our current implementation we can reduce the complexity of the decoding stages by leveraging a LUT to implement opcode maps described in [1]. We are working on including this table for mapping between opcode byte values and the instructions and operands they represent. In the other design choice, we can translate each x86 instruction into a sequence of micro-operations (uops), very similar to fixed length RISC instructions. These already translated micro-operation traces can be fetched from the simulator cache for further decoding and execution, thus removing the complex instruction decoder from the critical path.

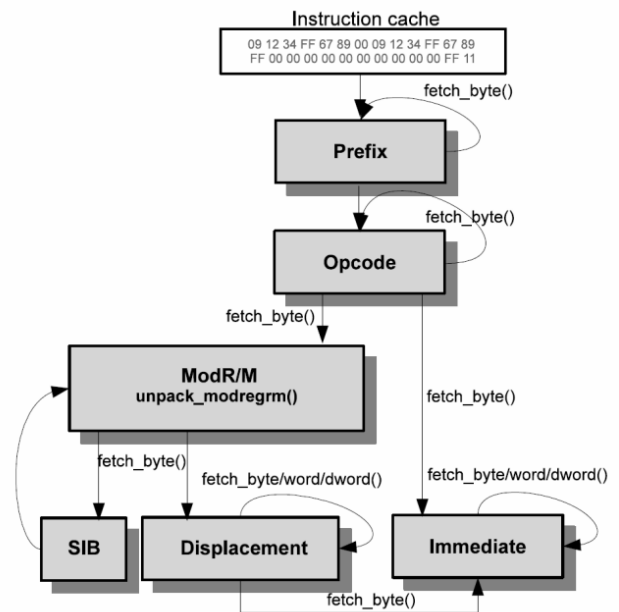


Figure 5. Instruction Decoding on x86 Architecture

3. EXPERIMENTAL RESULTS

In this section we present a set of experiments aimed at investigating the performance implications of different parallel program patterns on our simulation engine.

In this work we used an Nvidia GeForce GTX 295 graphics card, coupled with an Intel i7 CPU (@2.67 GHz) running Linux OS. The GeForce GTX 295 is a highly-parallel shared memory machine featuring two GTX200 GPUs on a single card, each of which has 30 Streaming Multiprocessors (SM) comprising 240 SPs with a total of 16K registers and connected through 938 MB device memory. For our experiments we only used one GPU card. As we already explained in various previous sections, the performance of our simulator is highly dependent on the parallel execution model adopted for the application being executed.

To investigate boundary cases to the achievable performance, we consider two synthetic patterns:

- **BC (Best case):** The target application exhibit data parallelism (SIMD), with parallel threads executing the same kernel on separate data subsets. Since all the simulated cores fetch the same target instructions, this case fully exploits the SIMT architecture of the GPU system.
- **WC (Worst case):** The target application employs task level parallelism (MIMD), where completely different tasks are assigned to parallel threads, and may operate on same or different sets of data. This setup is used to evaluate the performance overhead in a worst case scenario, where all the simulated cores execute different target instructions and hence diverge due to data dependent conditional branches.

Our ISS was designed for flexible future integration with other machine component simulators. We considered two basic design solutions suitable to the CUDA execution model for (future) design of a complex full-system simulator:

- **SK (Single Kernel):** The entire system simulator will be designed within a single CUDA kernel. In this case the various architectural components (ISS, cache, NOC) can be simulated in successive steps of a single function. This function can be off-loaded once from the host CPU onto the device and can execute until program completion. No other interactions with the host are required.
- **MK (Multiple Kernels):** Choosing this design option the cache, network and other system components will be modeled in separate CUDA

kernels. This requires that after each simulation step (i.e. a single instruction) every simulated core copies its current state from device memory to host memory. In case the instruction references memory, transactions are stored in a buffer (cache traces) which is also copied between host and device memory at every kernel swap. This design choice incurs a performance loss due to the extra time required for multiple kernel launches.

We expect other design variants to exhibit a performance level which is comprised within the described boundary cases. Similarly, we expect real application performance to sit somewhere between the above described worst and best case. To investigate the latter we consider in Sec. three realistic program kernels. The simulator performance is characterized by two speedup metrics.

- *Simulation Speedup:* The speedup achieved from parallelization of simulated cores on GPU hardware relative to sequentializing the simulation of different cores on a single serial processor (i.e. without all the CUDA-related processing).
- *Application Speedup:* The simulated program execution time with increasing number of simulated cores, normalized to the execution time of the program running on a single simulated core. Even if this metric is more related to the performance of the parallelization scheme rather than to the simulation performance, we found it useful to determine whether our simulation introduces significant overhead w.r.t. the expected speedup achievable with the given parallel algorithm.

We also provide simulation speed in MIPS which is calculated by target instructions executed per wall clock time of host.

The results are calculated for varying number of cores, in a range between 32 and 1024. Also three different execution configurations were considered using varying block sizes (32, 64, and 128). Block size in CUDA programming represents a set of concurrent threads that can cooperate and synchronize using shared memory private to that block.

3.1. Performance Comparison Under Corner Case Assumptions

In this section we define a set of different scenarios resulting from all possible combinations of the boundary cases identified above. These include simulator design

choices (**MK** - Different system components simulated within multiple CUDA kernels, **SK** - Full-system simulation takes place from within a single CUDA kernel) and parallel execution model (**WC** - MIMD parallelism, **BC** - SIMD parallelism). The set of considered configurations is summarized in Table 1.

Table 1. Different Scenarios.

	WC	BC
MK	Scenario 1	Scenario 2
SK	Scenario 3	Scenario 4

The best case for the ARM ISS has been simulated with all cores executing the same program, built by repeatedly looping over all the available instructions in the supported ISA. A total of 3500 instructions have been executed. In the worst case, the same program is loaded by every core, but each core is pointed to a different instruction where to start the program from. This ensures that all CUDA thread fetches a different instruction, and the whole execution is serialized.

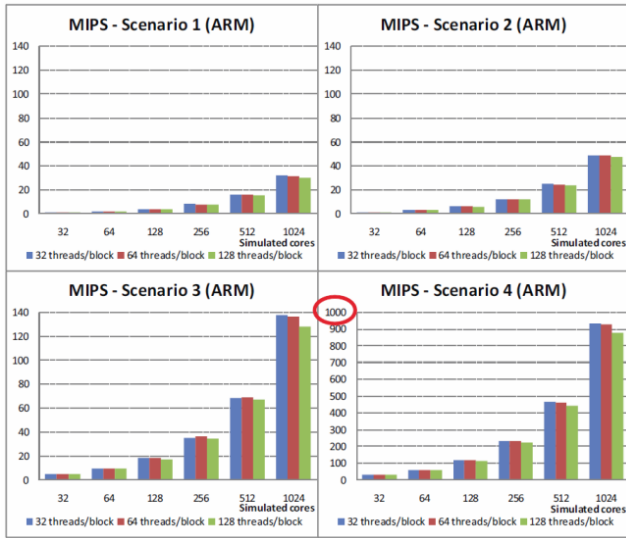


Figure 6.a. Simulated MIPS Under Different Scenarios for the ARM ISS

For the x86 ISS a similar setup has been considered, but due to the variable-length fetch/decoding stage we model best and worst case as follows. For the worst case we ensure that all available decoding schemes (i.e. lengths) are considered, thus assigning to different threads one of the possible combinations. For the best case every thread loads the same instruction, which is one that requires the minimum number of decoding steps.

In Fig. 6 we report the resulting MIPS for the described scenarios relative to both the ARM ISS (plot on the left)

and the x86 ISS (plot on the right). It is possible to notice an important difference between the ARM and the x86.

The most impacting factor in the ARM ISS implementation is the choice of a simulator design which relies on a few number of kernel launches. Indeed the worst case for single kernel implementation (scenario 3) is three times faster than the best case for multiple kernel implementation (scenario 2). The difference between MIMD and SIMD under multiple kernels (scenarios 1 and 2) is $\approx 2\times$, which grows up to $\approx 6,43\times$ under single kernel implementation. In terms of absolute numbers, the single kernel allows up to slightly less than 1 GIPS for SIMD applications (scenario 4, please note that this plot has a different Y scale), which is roughly one order of magnitude higher than the worst case (scenario 3, up to 140 MIPS). This gap is much less pronounced for the multiple kernel implementation, where SIMD application allow up to 50 MIPS and MIMD achieves 25 MIPS. SIMD applications (BC) under single kernel allow $18\times$ faster execution w.r.t. multiple kernels (scenario 4 vs scenario 2).

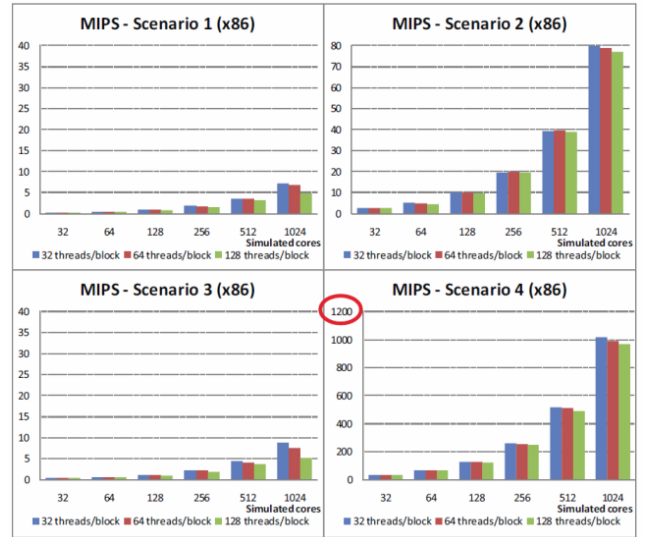


Figure 6.b. Simulated MIPS Under Different Scenarios for the x86 ISS

On the contrary, the performance of the x86 ISS implementation seems to be much more dependent on the execution pattern exhibited by the application. The gap between MIMD (WC) and SIMD (WC) is $\approx 11\times$ for multiple kernels and $\approx 100\times$ for single kernel, much higher than in the ARM. The worst case for single kernel implementation (scenario 3) is roughly nine times slower than the best case for multiple kernel implementation (scenario 2). This is because of the variable length decoding stage. In terms of absolute numbers the best case under single kernel allows up to ≈ 1 GIPS, slightly faster than the ARM. On the contrary both the worst cases, and

in particular that under single kernel (scenario 3) perform much worse than the ARM, due to the already discussed much more complex instruction decoding scheme.

3.2. Real Workloads

In this section we investigate the performance of three real-world program kernels, which are widely adopted in several applications both from the HPC and embedded domain, namely Matrix Multiplication, IDCT and FFT.

We adopt an OpenMP-like parallelization scheme to distribute work among available cores. More precisely, static loop parallelization is employed, where an identical number of consecutive iterations are assigned to parallel threads. This kind of parallelization relies on processor support to determine at runtime the physical ID of the thread/core onto which a given task (i.e. a chunk of loop iterations) is running. The dataset touched by each thread is differentiated based on the processor ID. Currently we are not supporting such a feature, so we take a different approach to simulate parallel execution.

Each thread executes the same loop, but reads different lower and upper bounds from stack-allocated variables. For the ARM processor, the ISS has been provided with a small routine which is in charge of managing the program's stack and heap. In this way we are capable of manually modifying the loop boundaries seen by each processor. The x86 ISS is still lacking this feature, so we manually substitute stack accesses with register read/write operations to achieve the same goal.

In Fig. 7 we show results for the speedup metrics described in the experimental setup, comparing the ARM and x86 ISS performance. It is possible to notice that for completely data parallel applications like Matrix Multiplication and IDCT, both the ARM and the x86 ISS allow ideal application speedup, getting very close to the best cases studied in the previous section. This applies to both the single kernel and multiple kernel cases, confirming that no significant overhead is added by our simulator. As for the simulation speedup, the single kernel implementation still achieves almost ideal results for both the ARM and the x86 ISS. Obviously this does not apply to the multiple kernel implementation, which adds a huge overhead for continuous kernel invocations and transfers between host and device memory. The multiple kernel implementation is slightly faster on the x86 ISS. The FFT benchmark deserves further discussion. Regarding the application speedup, it is possible to notice that the multiple kernel implementation on the ARM gets close to ideal results, whereas the x86 implementation only achieves $\approx 500\times$ speedup. This is due to a difference in the

actual parallelization schemes employed on the two simulators. In the code snippet below we show the simplified OpenMP code.

```
#define SIZE 1024

int fft[SIZE];
int steps=LOG2(SIZE)

for (j=0; j<steps; j++)
{
    int n=1;

    #pragma omp parallel for
    for (i=0; i<SIZE; i++)
    {
        if (i&n)
            exec_A ();
        else
            exec_B ();
    }

    n=n*2;
}
```

At the first outermost loop iteration even threads execute function exec A and odd threads execute function exec B. This implies the worst case performance in our simulator, since half of the threads in a warp are forced to stall when the other half is executing. Since, as already mentioned, the stack abstraction is not supported on the x86 ISS, all of the outer loops execute under this same pattern, thus always forcing worst case performance. On the contrary, the availability of the stack on the ARM processor allowed us to accurately model the kernel behavior even in the successive outer loop iterations. After the fourth iteration, functions exec A and exec B are grouped in chunks of 32 consecutive calls within a single thread. This leads to a situation in which all threads in a warp execute exec A, whereas function exec B is entirely executed by threads belonging to another warp. This allows exploiting parallelism over different multicores on the GPU, and thus exhibits much better performance. Regarding the single kernel implementation, the case with 1024 processors is slower w.r.t. the multiple kernel implementation. This happens because the number of innermost loop iterations in our FFT kernel is identical to the number of available processors. This implies that each processor executes a single invocation to either function exec A or function exec B, thus generating the same situation already discussed above for the worst case. The loss of performance implied by this particular configuration is significant when employing the single kernel implementation, whereas it goes almost completely unnoticed with the multiple kernel implementation, due to the higher overheads of the latter.

4. CONCLUSION

In this paper, we have presented a novel simulation infrastructure for massive parallel architectures which exploits the computational power of modern GPGPUs. Our experiments indicate that our approach can simulate

thousand of cores architecture providing fast simulation time and good scalability.

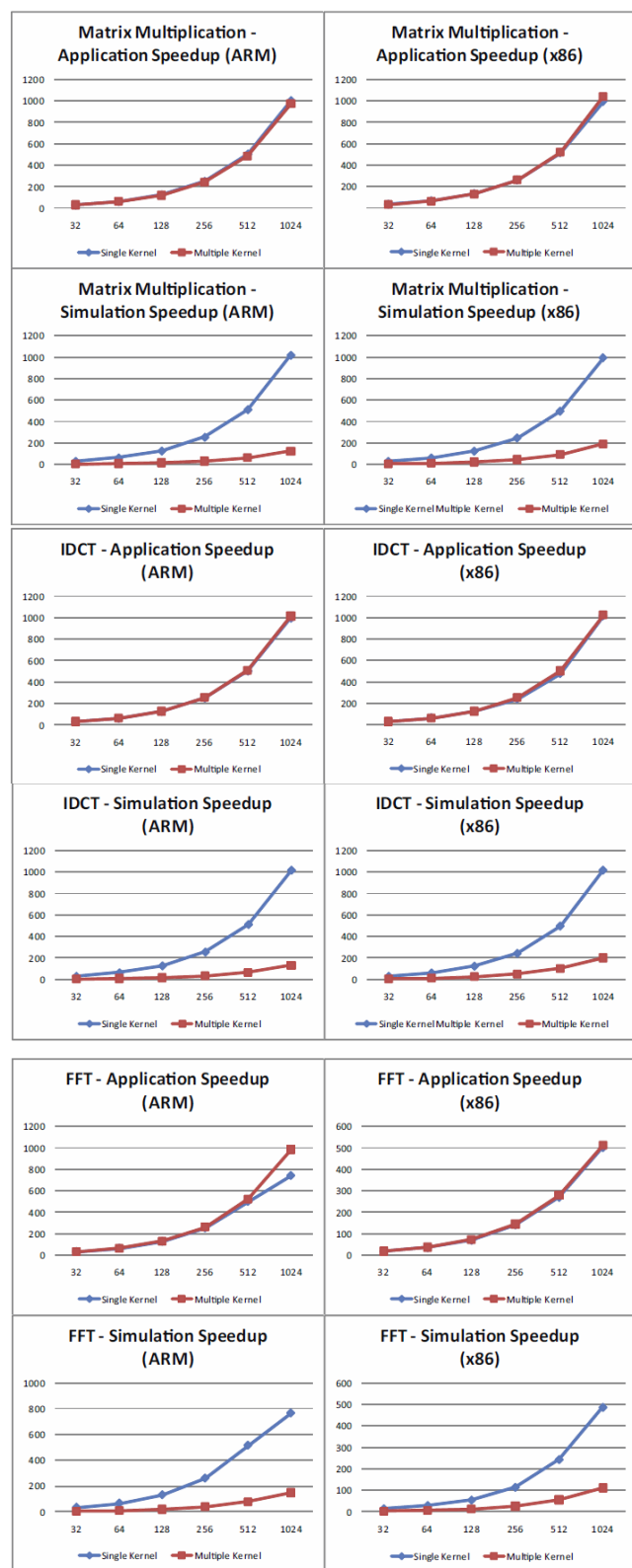


Figure 7. Speedup of Real Applications

ACKNOWLEDGEMENTS

The work described in this paper was partly supported by the PREDATOR Project funded by the European Community 7th Framework Programme, Contract FP7-ICT-216008, EC-FP7 STREP Project nr. 248776-PRO3D STREP, and the Swiss National Science Foundation under grant SNSF 200021-130048, Project nr. 130048.

REFERENCES

- [1] Intel 64 and ia-32 architectures SW developer's manual: <http://www.intel.com/products/processor/manuals/>.
- [2] E. Argollo, et al. "Cotson infrastructure for full system simulation." *Operating Systems Rev.*, v.43, 52–61, 2009.
- [3] K. Asanovic, et al. "A view of the parallel computing landscape." *Commun. ACM*, 52(10):56–67, 2009.
- [4] N. L. Binkert, et al. "The m5 simulator: Modeling networked systems." *IEEE Micro*, 2006.
- [5] S. Das, et al. "Gtw: A time warp system for shared memory multiprocessors." In *Proceedings of the 1994 Winter Simulation Conference*.
- [6] W. R. Davis, et al. "Demystifying 3d ics: The pros and cons of going vertical." *IEEE Design and Test of Computers*, 2005.
- [7] P. M. Dickens, et al. "A distributed memory lapse: Parallel simulation of message-passing programs." In *Workshop on Parallel and Distributed Simulation*, 1993.
- [8] K. Gulati and S. Khatri. "Towards acceleration of fault simulation using graphics processing units." In *DAC 2008*.
- [9] W. Han, et al. "Using gpu to accelerate cache simulation." In *Parallel and Distributed Processing with Applications*.
- [10] E. Lindholm, et al. "Nvidia tesla: A unified graphics and computing architecture." *IEEE Micro*, 2008.
- [11] S. Mukherjee. "Wisconsin wind tunnel ii: A fast and portable parallel architecture simulator," 1997.
- [12] S. Prakash and R. L. Bagrodia. "Mpi-sim: using parallel simulation to evaluate mpi programs." In *WSC '98*, 1998.
- [13] S. K. Reinhardt, et al. "The wisconsin wind tunnel: Virtual prototyping of parallel computers." *Sigmetrics Conference on Measurement and Modeling of Computer Systems*, '93.
- [14] J. Renau, et al. "SESC simulator," 2005. Available online at: <http://sesc.sourceforge.net>.
- [15] M. Rosenblum, et al. Complete computer system simulation: The simos approach. *IEEE Parallel Distrib. Technol.*, 1995.